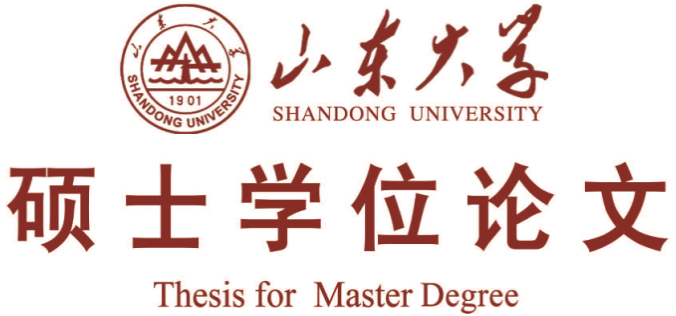
**分类号： 单位代码：10422**

**密 级： 学 号：**



（专业学位）

**论文题目：智能合约编译器差分测试中测试程序生成技术研究**

**作 者 姓 名 刘志浩**

**培 养 单 位 山东大学**

**专 业 名 称 计算机技术**

**指 导 教 师 余仲星**

**合 作 导 师**

**年 月 日**

原 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律责任由本人承担。

论文作者签名： 日 期：

关于学位论文使用授权的声明

本人同意学校保留或向国家有关部门或机构送交论文的印刷件和电子版，允许论文被查阅和借阅；本人授权山东大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其他复制手段保存论文和汇编本学位论文。

(保密论文在解密后应遵守此规定)

论文作者签名： 导师签名： 日 期：

# 摘要

智能合约是运行在区块链上的低级代码脚本。作为在区块链上运行的程序，智能合约可以在不依赖可信第三方的情况下促成两个当事方之间的合约。智能合约在区块链中具有广泛的应用前景，可以在各种领域实现自动化、透明化和安全化的交易。随着区块链技术的不断发展和智能合约的进一步成熟，相信智能合约将会在未来发挥越来越重要的作用。

为了保证智能合约能够正确编译，要保证其编译器的准确性尤为关键。编译器运行过程中不应该崩溃，也不应该出现错误的编译结果。实际上，任何运行在计算机上的软件都是由编译器或类似编译器的工具处理的。因为编译器是构建软件的关键基础设施，因此编译的正确性至关重要。为了验证和提高编译器的正确性，人们投入了大量的研究工作来测试编译器。

测试编译器离不开测试用例的生成，从编译器测试的早期开始，就一直使用手动构建的测试程序。这样的测试程序因为可以根据特定的需求进行调整，因此可以有效的发现编译器的问题。尽管手动构建测试程序十分有效，但是手动编写测试程序需要大量的工作。因此人们一直努力为测试编译器设计测试程序生成器。

本文通过Solsmith生成Solidity代码来测试编译器。Solsmith是一个专门为发现Solidity编译器问题而设计的工具，其通过生成Solidity代码来测试编译器的正确性和性能。Solsmith的代码生成流程包括创建测试环境、定义状态变量、动态数组、常量变量、修饰器、事件语句以及构建函数框架。通过这些步骤，Solsmith能够生成包含多种控制结构（如赋值语句、条件语句、循环语句和内联汇编语句）的智能合约代码。

通过采用一定生成策略，Solsmith可以生成更多样化且具有更高测试效率的智能合约代码，有助于发现和解决编译器优化器可能存在的问题，从而提高智能合约的安全性和稳定性。

为了验证SolSmith的有效性，我们使用SolSmith生成的代码分别在三种不同配置的Solidity编译器上运行，进行了实验验证，以得出SolSmith的有效性。实验结果表明，SolSmith能够有效的进行Solidity编译器的测试。

**关键词：**编译器测试；智能合约；程序生成；未定义行为

# ABSTRACT

Smart contracts are low-level code scripts running on the blockchain. As programs running on the blockchain, smart contracts can facilitate contracts between two parties without relying on trusted third parties. Smart contracts have broad application prospects in the blockchain, enabling automation, transparency, and security in transactions across various domains. With the continuous development of blockchain technology and the further maturation of smart contracts, it is believed that smart contracts will play an increasingly important role in the future.

To ensure that smart contracts can be compiled correctly, it is crucial to ensure the accuracy of their compilers. During the compilation process, the compiler should not crash or produce erroneous compiled results. In fact, any software running on a computer is processed by a compiler or similar tools. Because the compiler is a crucial infrastructure for building software, the correctness of compilation is essential. To verify and enhance the correctness of compilers, extensive research efforts have been devoted to testing compilers.

Testing compilers relies on the generation of test cases. From the early stages of compiler testing, manually constructed test programs have been used. Such test programs can effectively identify compiler issues because they can be adjusted according to specific needs. Although manually constructing test programs is very effective, it requires a lot of work. Therefore, efforts have been made to design test program generators for testing compilers.

This paper utilizes Solsmith to generate Solidity code for testing the compiler. Solsmith is a tool specifically designed to discover issues in the Solidity compiler by generating Solidity code to test the correctness and performance of the compiler. The code generation process of Solsmith includes creating a testing environment, defining state variables, dynamic arrays, constant variables, modifiers, event statements, and constructing function frameworks. Through these steps, Solsmith can generate smart contract code containing various control structures (such as assignment statements, conditional statements, loop statements, and inline assembly statements).

By adopting certain generation strategies, Solsmith can generate smart contract code with higher diversity and testing efficiency, helping to discover and address potential issues in the compiler optimizer, thereby improving the security and stability of smart contracts.

To validate the effectiveness of SolSmith, we conducted experimental verification by running the code generated by SolSmith on three different configurations of Solidity compilers. The experimental results demonstrate that SolSmith can effectively test Solidity compiler.

Keyword**:** compiler testing; smart contract; program generation; undefined behavior

目录

[摘要 I](#_Toc158888010)

[ABSTRACT II](#_Toc158888011)

[1 绪论 1](#_Toc158888012)

[1.1 研究目的与背景 1](#_Toc158888013)

[1.2 国内外研究现状 2](#_Toc158888014)

[1.2.1 测试程序生成技术 2](#_Toc158888015)

[1.2.2 Solidity编译器测试 4](#_Toc158888016)

[1.3 本文研究内容和组织结构 4](#_Toc158888017)

[2 相关技术与研究 5](#_Toc158888018)

[2.1 编译器测试 5](#_Toc158888019)

[2.2 测试程序生成技术 7](#_Toc158888020)

[2.3 未定义行为 8](#_Toc158888021)

[2.4 差分测试 9](#_Toc158888022)

[2.5 本章小结 10](#_Toc158888023)

[3 测试程序生成技术 12](#_Toc158888024)

[3.1 语法导向 12](#_Toc158888025)

[3.1.1词缀语法 12](#_Toc158888026)

[3.1.2 属性语法 13](#_Toc158888027)

[3.1.3 w语法 14](#_Toc158888028)

[3.2 语法辅助 15](#_Toc158888029)

[3.3 其他方法 16](#_Toc158888030)

[3.4 本章小结 19](#_Toc158888031)

[4 SolSmith的实现 20](#_Toc158888032)

[4.1 生成代码 20](#_Toc158888033)

[4.1.1 创建环境 20](#_Toc158888034)

[4.1.2 创建函数框架 20](#_Toc158888035)

[4.1.3 创建语句 21](#_Toc158888036)

[4.2 生成策略 22](#_Toc158888037)

[4.2.1 频繁启用优化 22](#_Toc158888038)

[4.2.2 避免中间表征导致的区别 25](#_Toc158888039)

[5 实验结果与分析 28](#_Toc158888040)

[5.1实验环境 28](#_Toc158888041)

[5.2 实验过程 28](#_Toc158888042)

[5.3 实验结果 30](#_Toc158888043)

[5.4 本章小结 30](#_Toc158888044)

[结论 31](#_Toc158888045)

[参考文献 32](#_Toc158888046)

[致谢 38](#_Toc158888047)

**CONTENTS**

[Chinese abstract I](#_Toc158883789)

[English abstract II](#_Toc158883790)

[Chapter1 Introduction 1](#_Toc158883791)

[1.1 Background and purpose of research 1](#_Toc158883792)

[1.2 Analysis of research status at home and abroad 2](#_Toc158883793)

[1.2.1 Test Program Generation 2](#_Toc158883794)

[1.2.2 Solidity compiler testing 4](#_Toc158883795)

[1.3 This paper studies the content and organization structure 4](#_Toc158883796)

[Chapter 2 Related research and technologies 5](#_Toc158883797)

[2.1 Compiler testing 5](#_Toc158883798)

[2.2 Test Program Generation 6](#_Toc158883799)

[2.3 Undefined behaviour 8](#_Toc158883800)

[2.4 Differential testing 9](#_Toc158883801)

[2.5 Summary of this chapter 10](#_Toc158883802)

[Chapter 3 Test Program Generation 12](#_Toc158883803)

[3.1 Grammar-directed Approaches 12](#_Toc158883804)

[3.1.1 Affix grammar 12](#_Toc158883805)

[3.1.2 Attribute grammar 13](#_Toc158883806)

[3.1.3 W gammar 14](#_Toc158883807)

[3.2 Grammar-aided Approaches 15](#_Toc158883808)

[3.3 Other Approaches 16](#_Toc158883809)

[3.4 Summary of this chapter 19](#_Toc158883810)

[Chapter 4 Implementation of SolSmith 20](#_Toc158883811)

[4.1 Generating code 20](#_Toc158883812)

[4.1.1 Creating the environment 20](#_Toc158883813)

[4.1.2 Creating function 20](#_Toc158883814)

[4.1.3 Creating statement 21](#_Toc158883815)

[4.2 Generation policies 22](#_Toc158883816)

[4.2.1 Frequent triggering of the optimizer 22](#_Toc158883817)

[4.2.2 Avoid IR\_based codegen changes 25](#_Toc158883818)

[Chapter 5 Experimental results and analysis 28](#_Toc158883819)

[5.1 Experimental environment 28](#_Toc158883820)

[5.2 Experimental process 28](#_Toc158883821)

[5.3 Experimental result 30](#_Toc158883822)

[5.4 Summary of this chapter 30](#_Toc158883823)

[Conclusion 31](#_Toc158883824)

[Reference 32](#_Toc158883825)

[Acknowledgement 38](#_Toc158883826)

# 1 绪论

## 1.1 研究目的与背景

“智能合约”一词由尼克·萨博在1990年代中期首次提出[1]。他建议将合同条款转化为代码，并将其嵌入到软件或硬件中，使其能够自我执行，以最小化交易各方之间的合同成本，并避免在合同执行过程中出现意外异常或恶意行为。

目前，不同学科的人们对“智能合约”一词的使用方式不尽相同。一些人将可以由软件表示（或者至少其中部分可以由软件表示）的法律合同称为“智能合约”。而另一些人则将“智能合约”视为一种代码脚本，设计用于一旦满足预定义条件就执行特定任务的脚本；这些脚本通常（虽然不一定）在分布式账本（例如区块链）上运行[2]。Clack等人提出了一个广泛涵盖上述定义范围的智能合约定义。他们将智能合约定义为“可自动化执行的协议。通过计算机自动化执行，尽管某些部分可能需要人为输入和控制。可以通过法律强制执行权利和义务或通过不可篡改的计算机代码执行来执行”[3]。

在本文中，我们主要关注运行在区块链上的低级代码脚本。作为在区块链上运行的程序，智能合约可以在不依赖可信第三方的情况下促成两个当事方之间的合约。从技术上讲，智能合约是一个包含数据（例如账户余额）和可执行代码的程序。智能合约可以存储在区块链上，并且可以在满足某些预条件时自动执行。在每次执行智能合约后，其状态可以在区块链上更新[4]。

在区块链中，智能合约具有广泛的应用，其中最显著的应用之一是加密货币交易。智能合约可以用来执行加密货币之间的交易，确保交易的安全性和可靠性，而无需依赖于传统的金融机构。此外，智能合约还可以应用于众多领域，如供应链管理、物联网、数字身份认证、票据结算等。

智能合约在供应链管理中的应用是另一个重要应用。通过在区块链上部署智能合约，可以实现供应链中的自动化执行和监控。例如，当产品从供应商处发货时，智能合约可以自动释放付款给供应商，同时更新产品的状态和位置信息。这种自动化的流程不仅提高了供应链的效率，还减少了人为错误和欺诈行为的可能性。

另一个重要的应用领域是数字身份认证。通过在区块链上创建智能合约，可以实现安全的身份验证机制，避免个人信息被盗用或篡改。智能合约可以存储用户的身份信息，并在需要时验证用户的身份，从而确保交易的安全性和可靠性。

总的来说，智能合约在区块链中具有广泛的应用前景，可以在各种领域实现自动化、透明化和安全化的交易。随着区块链技术的不断发展和智能合约的进一步成熟，相信智能合约将会在未来发挥越来越重要的作用。

为了保证智能合约能够正确编译，要保证其编译器的准确性尤为关键。在软件开发过程中编译器是重要的工具，因为它是构建其他软件的核心部分。几乎所有要在计算机上运行的程序，从操作系统到小脚本，都是由编译器或类似编译器的工具来处理的。因为编译器是软件基础的核心部分，所以它分布的十分广泛。例如，广泛使用的编程语言（如用于C的GCC）的流行编译器由数百万用户运行。除了这些编译器的直接用户（通常是开发人员）之外，更多的人在执行编译程序时间接地依赖编译器。尽管已经做出了相当大的努力来提高编译器的质量，就像所有其他软件一样，编译器仍然存在 bug[5]。编译器错误可能导致从正确的源代码生成不正确的二进制代码。

此外，编译器中的一个 bug 可能会传播到构建在它上面的任何应用程序，并导致令人惊讶的、可能有害的错误行为。例如，Java7实现中的一个错误编译错误导致几个流行的 Apache 项目崩溃。有时，编译器错误甚至可能被故意注入，以破坏已编译应用程序的安全性。例如，Apple 的 Xcode 开发环境的一个恶意变体包含一个编译器问题，它在每个已编译的应用程序中引入了一个后门。这样的编译器后门还可以利用偶然引入的 bug，通过 LLVM 中公开的 bug 破坏 Unix sudo 工具就可以证明这一点[6]。编译器错误不仅会导致意想不到的行为，并可能带来严重的后果，而且还会使软件调试更加困难。原因是开发人员很难确定软件故障是由他们正在开发的程序还是他们正在使用的编译器造成的[7]。如果当有错误的编译器将正确的程序优化为具有不正确运行时行为的可执行程序时，程序开发人员并不清楚是什么导致了意外行为。由于应用程序开发人员通常认为错误行为往往是由他们自己引入的错误造成的，他们可能会花费很长时间才最终认识到编译器错误是根本原因。

## 1.2 国内外研究现状

本节主要介绍测试程序生成技术，Solidity编译器测试研究的国内外现状。

### 1.2.1 测试程序生成技术

目前对于测试程序生产技术国内外对于测试程序生成技术研究比较多，但是大多集中在主流语言反面。

Purdom提出了一种基于上下文无关语法来测试解析器和语法正确性的方法[8]。主要焦点是验证从无关上下文的语法自动生成的解析器，并查找未简化的语法(其中语法包含不能用于句子派生的符号)。生成从一个唯一的开始符号开始，并通过应用语法中的左右重写规则继续进行。该方法使用一些启发式方法通过递归地遍历语法来生成短句。对于测试解析器，通过生成的程序涵盖解析器的不同状态和转换。Purdom在LALR(1)解析器上评估了他的方法，并表明在许多情况下，生成的测试程序能够覆盖解析器的多个状态和转换。

Purdom基于上下文无关语法的测试程序生成在测试编译器的所有部分时存在缺陷。如果只使用上下文无关的语法，就很难表达语言的上下文敏感特性。由于Purdom的最终目标是测试不期望语义正确的程序的解析器，因此这个缺点并不影响该方法。

Yang等人提出了一种名为Csmith的语法辅助工具，用于创建C测试程序[9]。该工具基于Randprog[10]。除了Randprog中存在的函数、全局和局部变量、const和volatile变量之外，Csmith还生成了包含控制流语句、结构体、数组和大多数C表达式的程序。该方法使用复杂的启发式方法来避免生成具有未定义行为以及依赖于未指定行为的C程序。生成过程首先创建一个main函数和一组结构类型声明，其中每个声明都包含随机决定类型的随机数量的成员变量。使用main函数作为起点，其余的C代码是基于C语法的子集生成的。根据生成的当前状态，Csmith根据概率表和过滤函数选择是否生成。过滤器函数增强了上下文敏感性。在生成过程中，Csmith执行某些安全检查，并仅在所有安全检查通过时创建代码片段。

当然，并不是所有自动测试程序生成的方法都是建立在目标编程语言的语法之上的。下面是一组不直接使用语法作为输入的方法。

Berry提出了基于真实程序员使用语言特性的频率为编译器生成测试程序[11]。Mandl是一种避免在程序中产生重复元素的方法。它使用一种称为正交拉丁平方的独特方法来生成用于验证Ada编译器的测试程序[12]。Randprog生成随机的C程序来查找C的volatile限定符的错误编译。Randprog生成的每个程序都是硬编码的，首先包含一些随机初始化的全局变量，这些全局变量要么是const，要么是volatile。然后程序包含声明局部变量和包含使用全局变量和局部变量的表达式的函数。最后，为了使程序可执行，每个程序都包含一个main函数。

### 1.2.2 Solidity编译器测试

尽管有一些研究致力于分析Solidity编写的智能合约中的错误，但是很少有研究专注于测试Solidity编译器的错误。

AFL 编译器-fuzzer使用基于文本的变异来测试不同的编译器，包括 Solidity编译器。但是AFL是一个比较通用的用来测试编译器的fuzzer，没有考虑语言的特殊性。

FUZZOL是第一个用于系统地测试Solidity编译器的安全性和可靠性的语法感知的变异fuzzer[13]。

目前在Solidity语言上还没有专门的随机测试程序生成工具。

## 1.3 本文研究内容和组织结构

论文的结构如下：

第一章主要介绍本文的研究背景和研究意义。主要介绍了智能合约，测试程序生成技术、Solidity编译器测试的现状，介绍了本文的主要内容和论文结构；

第二章主要介绍了与本文相关的技术内容，包括编译器测试的重难点，差分测试常用的形式，测试程序生成技术的基础内容；

第三章详细介绍了SolSmith的具体实现。包括工具的实现原理、整个工具的具体流程和各个部分的实现细节；

第四章介绍了实验对象、实验过程和结果分析；

第五章是本片论文的总结。对本文内容进行归纳总结，描述了工具现有的不足和未来的展望。

# 2 相关技术与研究

本章主要介绍了本文中使用到的相关技术，主要包括编译器测试，测试程序生成技术，未定义行为，差分测试的原理等内容。

## 2.1 编译器测试

考虑到编译器的重要性，正确地实现它们是非常重要的。然而，达到这个目标并非易事，因为编译器是复杂的软件。典型的编译器由一系列相互作用的组件组成，这些组件处理源语言的词法分析、将源语言解析为中间表示、应用语义检查、优化代码以及生成目标语言代码。目前，并行性和面向对象特性的优化阶段和实现使编译器更加复杂。由于这种复杂性，传统的计算机科学课程通常至少要花一整个课程来介绍构造编译器的艺术。与其他复杂的软件相比，编译器处理的领域更加丰富。编译器的输入和输出通常都是用图灵完备语言编写的程序。此外，输入域和输出域都是无限大的，因为程序可以任意长。因此，正确实现编译器是十分困难的[14]。

正确实现编译器的困难给测试实现带来了一些挑战。一个挑战是缺乏关于编译器应该做什么的正式规范。虽然高级规范是隐式已知的(编译器以保持语义的方式将源程序转换为目标程序)，但低级细节通常是不指定的。例如，很少指定何时应用哪些优化，这使得很难检查编译器是否应用了所有所需的优化。例如，LLVM编译器有58个优化转换通道，它们可以以各种方式组合在一起，但是没有规范说明何时使用这些通道中的哪一个。

测试的另一个挑战是编译器处理的输入和输出语言的语义丰富性。由于源代码可以广泛的表达，包括许多导致程序崩溃的计算，因此很难自动创建具有重要行为的源代码。然而，这种重要的程序通常需要深入到编译器的工作流程中，并测试难以触及的代码。例如，输入程序必须通过词法分析器和解析器中的所有完整性检查和类型检查，然后才能测试编译器执行的优化。另一方面，非编译器程序通常没有这种复杂的检查。并且，输入的微小变化可能导致预期行为的巨大差异。例如，更改输入程序中的单个字符可能会导致编译器选择完全不同的路径并暴露完全不同的行为。

第三个重要的挑战是编译器有各种各样的选项和特性。例如，大多数编译器提供不同的优化级别，支持源语言的几种变体，并考虑多个目标平台。这种大量的配置创造了一个难以彻底探索的巨大空间，并且与常规输入空间正交。

当然，除了这些挑战之外，编译器还有一些属性可以简化验证其正确性的问题。其中一个特性是编译器的输入是用编程语言编写的，也就是说，可能输入的空间是由语言语法明确定义的。另一个简化编译器测试的特性是源语言的语义通常被指定，不是在语言规范中非正式地指定，就是在编程语言的核心中正式指定。因此，在运行程序时，(大多数)程序的预期行为以及编译器输出都是已知的。最后，还有一个可以简化验证编译器的特性，即对于大多数流行的编程语言，存在多个假定的等效实现，编译器测试可以利用这些实现作为差分测试的oracle[15]。

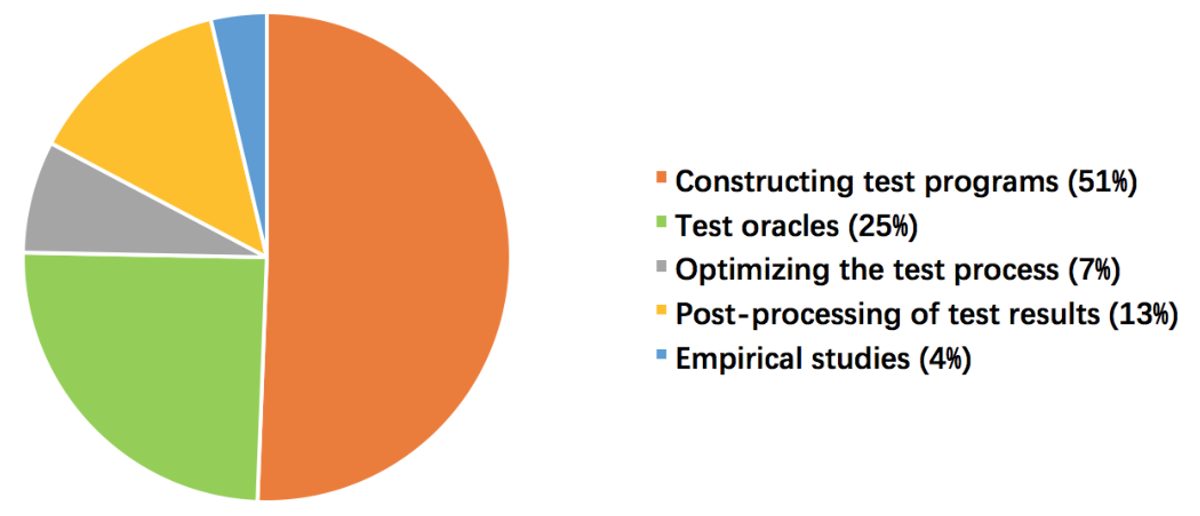
由于编译器的重要性以及在提高其可靠性和正确性方面所涉及的有趣挑战，该领域受到了研究人员和实践者的极大关注。图2-1显示了1970年-2018年间85篇相关论文的研究角度。包括构建测试程序、测试预言机、优化测试过程、测试结果的后处理和实证研究[14]。

图2-1编译器测试研究角度占比

与一般的软件测试一样，大多数关于编译器测试的论文也关注于构建测试程序和测试oracle。构建测试程序尤其吸引了大多数人的关注。这是意料之内的，因为构建测试程序是测试过程的初始步骤(只有在构建了测试程序之后，测试过程才能开始)。由于研究人员的努力，编译器测试取得了巨大的成功，它在最近几年取得了巨大的进步。在过去的十年中，已经开发了几种广泛采用的工具和方法，其中一些在流行语言的编译器中发现了数百个错误。

## 2.2 测试程序生成技术

任何类型的软件测试都需要测试用例。在编译器测试领域，测试程序构成测试用例的一部分。为编译器构建测试程序不是一件简单的事情。主要是因为很难构建有效的、多样化的、满足特定需求的测试程序。

构造有效的程序是很重要的，因为某些语言有限制，只能以某种方式或在某种上下文中使用某些语言结构。构造无效的程序对于测试编译器的用处有限，因为一个程序要经过编译器的多个处理阶段;如果向编译器提供无效的输入程序，那么该程序往往会在处理的初始阶段被丢弃。例如，如果构造的JavaScript程序包含一个不在函数内的return语句，则该程序会被视为语法无效。这样的程序不会经过JavaScript引擎的代码生成或优化阶段。例如，在C语言中，生成具有未定义行为的程序不能被视为有效的测试程序，因为结果始终是正确的。同样地，对于许多语言来说，在程序中使用之前必须先声明标识符。在保持这些限制的情况下构建程序并不总是直接的。特别是对于静态类型语言来说，在测试程序构建过程中保持类型约束并不容易，这经常导致构建无效的测试程序[14]。

测试程序的多样性。与所有测试输入一样，构建的测试程序应该是多种多样的。一组语法不同的测试程序将运行编译器的不同部分，并可能增加被测编译器的代码覆盖率。这可能有助于发现bug。在测试编译器领域，已经提出了多样性指标，如测试程序之间的距离[16]。虽然构建测试程序的多样性是很必要的，但是很难构建一个多样化的测试程序集，因为语言语法结构数量的增加会对生成的程序的有效性产生不利影响。例如，一组构造好的C测试程序可能需要包含使用所有可用类型的变量及其相应的操作。但是，对于测试程序构造函数来说，这样做并不是很简单。随着更多特性的使用，会更容易生成无效的测试程序。

满足需求的测试程序。当某些测试方法施加限制时，测试程序的构建也变得困难。例如，常用的测试方法是用两个编译器编译程序并比较执行结果。在这种情况下，输入程序应该没有未定义的行为。另一方面，如果我们只是测试编译器崩溃，那么输入程序就不要求一定没有未定义的行为。除了有效性和多样性需求之外，构建遵循这些限制或一致性的测试程序就十分具有挑战性了。

从编译器测试的早期开始，就一直使用手动构建的测试程序。这样的测试程序因为可以根据特定的需求进行调整，因此可以有效的发现编译器的问题。尽管手动构建测试程序十分有效，但是手动编写测试程序需要大量的工作。因此人们一直努力为测试编译器设计测试程序生成器。主要可以分为三种方向：语法导向、语法辅助和其他方法。本文会在第三章中对这三种方向进行详细介绍。

## 2.3 未定义行为

在系统编程语言的设计中，赋予编译器多大的自由来为目标指令集生成有效的代码是非常难权衡的。一方面，程序员希望程序在所有硬件平台上的行为都相同。另一方面，程序员希望通过允许编译器利用其硬件平台的指令集的特定属性来获得高性能。语言用来进行这种权衡的一种技术是将某些程序结构标记为未定义的行为，未定义行为对编译器编写者没有任何要求[17]。

以C语言中的未定义行为举一个例子，如以零为除数的整数除法。相应的机器指令在x86上导致硬件异常，而PowerPC会静默地忽略它。C语言没有在指令集之间强制统一语义，而是将除零定义为未定义的行为，允许C编译器为目标平台选择有效的实现[18]。

对于这个特定的例子，当C程序除零时，编译器编写者不会被迫产生异常，这允许PowerPC的C编译器使用不产生异常的指令。如果C语言坚持使用除零的异常，那么C编译器将不得不加入额外的指令来检测PowerPC上的除零。

一些语言(如C/C++)将许多结构定义为未定义行为，而其他语言(如Java)的未定义行为较少[19]。但是，在Java等高级语言中存在未定义的行为表明，这种权衡不仅限于低级系统语言。

C编译器相信程序员不会提交具有未定义行为的代码，并在此假设下优化代码。对于不小心使用具有未定义行为的结构的程序员来说，这可能会导致意外的程序行为，因为编译器可能会删除代码(例如，删除访问控制检查)或以程序员没有预料到的方式重写代码。

因此我们在生成测试用例的时候是不能出现未定义行为的。如果我们生成的程序中有未定义行为，即使编译器出现了与我们预期不符的结果，我们也不能就此判定编译器是有问题的，所以未定义行为是一定要避免的。常见的避免未定义行为的方法有：从语法上避免未定义行为，插入动态检查，使用检查工具检查生成程序中是否有未定义行为，静态分析。

从语法上避免未定义行为，如为了避免出现指针内容上的未定义行为，在测试程序生成过程中不加入指针类型。从根本上避免指针方面的未定义行为。Quest [20]使用了这种方法，但是这种方法极大的限制了生成代码的表达性。

生成没有静态约束的代码，根据需要插入动态检查以防止未定义的行为。Csmith [9]使用了这种方法，这种方法通过模糊无处不在的安全检查背后潜在的可优化模式而牺牲了表达性。

生成程序时不必担心未定义的行为，然后使用检查工具丢弃那些未定义的行为。当生成的程序非常小时，这种方法效果很好，但Solidity语言现在并没有很好的针对未定义行为检查的工具。

运行静态分析，以保守地避免生成过程中未定义的行为。Csmith将此策略用于指针相关的UBs和Orange系列生成器[21]。

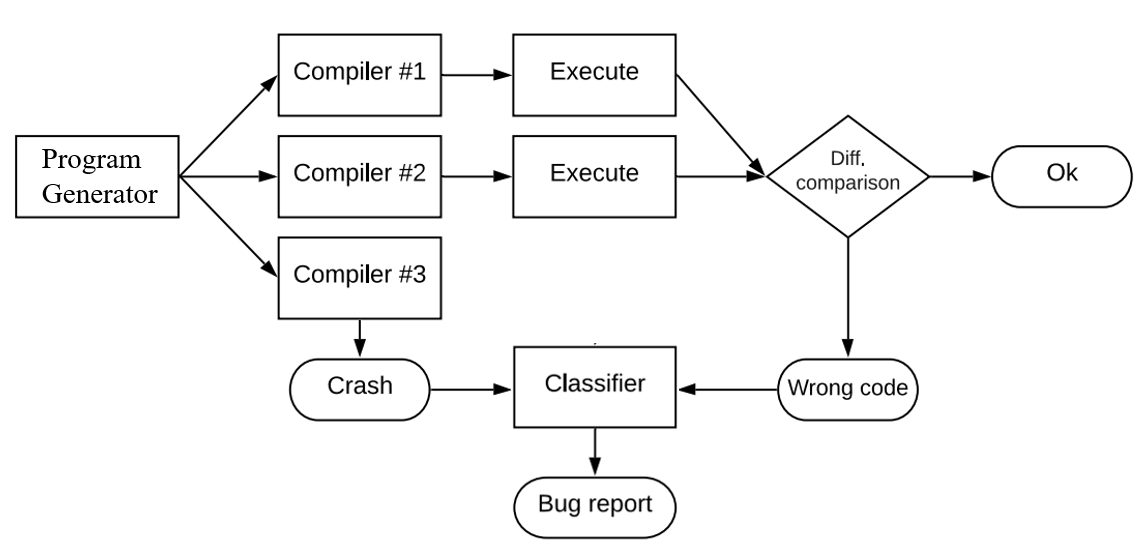
## 2.4 差分测试

与任何测试活动一样，编译器测试必须解决test-oracle问题，即确定给定的测试程序是否暴露了任何不期望的行为。为了应对这一挑战，目前提出的解决方法主要分为两类:差分测试[15]和变质测试[22]。为了解决编译器等复杂软件的test-oracle问题，McKeeman等提出了差分测试的概念。通常，编译器的差异测试至少需要两个基于相同规范设计和实现的编译器，然后比较这些可比编译器的结果，以确定是否检测到编译器错误。为了选择要比较的实现，有几种不同的编译器差异测试。在表2-1中总结了编译器测试中三种广泛使用的差异测试策略的用法。

表2-1 三种差分测试策略在编译器测试中的应用

|  |  |  |  |
| --- | --- | --- | --- |
| 论文 | 交叉编译 | 交叉优化 | 跨版本 |
| Sheridan  Ofenbeck et al.  Sassa and Sudosa  Morisset et al.  Le et al.  Béra et al.  Hawblitzel et al.  Chen et al.  Sun et al. | 🗸  🗸  🗸  🗸 | 🗸  🗸  🗸  🗸  🗸  🗸 | 🗸  🗸  🗸 |

差分测试编译器框架流程如图2-2所示，将生成的测试程序，分别放入多个不同配置的编译器，运行测试程序，如果不同配置的编译器得出的结果不同，那么说明某个或某些编译器在运行时出现了错误，如果编译器运行崩溃了说明该配置的编译器存在问题。

图2-2 差分测试编译器框架流程图

交叉编译策略:通过比较不同编译器产生的结果来检测编译器错误。这个策略是编译器差分测试中最通用的概念。首先，对于相同的编程语言，不同的编译器应该对相同的输入产生相同的输出。其次，如果输入可以触发错误，由于实现之间的差异，不同的编译器很少在相同的输入下暴露相同的错误并产生相同的错误输出。因此，如果两个编译器在相同的输入下产生不同的输出，其中一个必须包含错误。

交叉优化策略:通过比较在单个编译器中实现的不同优化产生的结果来检测编译器错误。该策略是目前编译器测试研究中应用最广泛的策略。对于同一个编译器不同的优化等级，应该对相同的输入产生相同的输出。但是如果同一个编译器的不同优化等级在相同的输入下产生了不同的输出，那么必定在某个优化等级下编译器出现了错误。

跨版本策略:通过比较单个编译器的不同版本产生的结果来检测编译器错误。从表2-1中我们可以发现，目前并没有研究人员单独使用该策略来测试编译器中的错误。

## 2.5 本章小结

本章主要介绍了与本文相关的一些技术的研究。在2.1节介绍了在编译器测试常见的几个研究方向和编译器测试过程中常遇到的一些挑战；在2.2节中介绍了测试程序生成技术，包括生成测试程序时需要重点关注的内容和常用的生成方法；在2.3节中介绍了未定义行为，为什么要在生成测试程序的时候避免未定义行为，以及常见的测试程序生成器如何避免未定义行为；2.4节介绍了差分测试，差分测试能够解决哪些问题，以及常用的差分测试的方法有哪些。

# 3 测试程序生成技术

本章主要对第二章中提到的常见的测试程序生成技术的三个方向：语法导向，语法辅助，其他方法进行详细介绍。

## 3.1 语法导向

语法导向的测试程序生成方法采用语言语法作为输入，并基于该语法生成程序。语法导向的方法是为测试编译器自动生成测试程序而提出的第一种方法。给定语言的上下文无关语法，遍历并生成该语言的字符串。为此，Purdom[8]提出了一种基于上下文无关语法来测试解析器和语法正确性的方法。主要目标是验证无关上下文的语法的解析器，并查找未简化的语法(其中语法包含不能用于句子派生的符号)。生成从一个唯一的开始符号开始，并通过应用语法中的左右重写规则继续进行。该方法使用一些启发式方法通过递归地遍历语法来生成短句。对于测试解析器，希望生成的程序涵盖解析器的不同状态和转换。Purdom在LALR(1)解析器上评估了他的方法，并表明在许多情况下，生成的测试程序能够覆盖解析器的多个状态和转换。

侧重于测试编译器中难以触及的部分的方法使用不同的方法来解决测试程序生成过程中的上下文敏感性问题。基于语言语法生成可编译程序的最初尝试倾向于用上下文敏感的特性扩展与上下文无关的程序。扩展语法形成了一个被称为两级语法的语法族，由Adriaan van Wijngaarden引入，用于指定ALGOL 68[23]。为测试编译器而生成的测试程序使用三种特定的两级语法，即w语法、属性语法和词缀语法。下面简要介绍两级语法的这三种变体，并解释使用每种变体的方法。

### 3.1.1词缀语法

词缀语法最初是为语言学应用而发明的[24]，它是一种使用词缀对语法产物进行上下文敏感概念扩展上下文无关语法的方法。原则上，我们后面介绍的词缀语法和属性语法的区别仅在于它们的形式表示法[25]。两者都使用参数扩展与上下文无关的语法。

受词缀语法的启发，Hanford[26]提出了一种使用扩展的上下文无关语法的方法。扩展语法的规则与其他已知的语法生成器附加在一起。语法生成器有效地充当存储库。如图3-1所示的简化的例子。

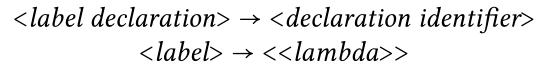
图3-1词缀语法扩展示意

图3-1中的第一行是语法规则，而第二行是语法生成器。这些规则意味着每当声明标识符(例如a)时，就会激活语法生成器，并将名为→a的规则添加到无关上下文的语法中。这种将规则添加到与上下文无关的语法中，有效地生成具有某些上下文敏感特性的程序，例如，“声明后使用”。对于语法中的每个非终结符，将做出伪随机决定，选择另一个非终结符或一个终结符。Hanford将该方法实现为一个称为语法机的程序，该程序为PL/I子集生成语法上有效的测试程序。

另一种使用词缀语法的方法是由Houssais[27]提出的，他使用这些语法生成测试ALGOL 68实现的程序。这种方法将词缀的域限制为整数。程序生成器通过分别枚举每个程序，然后将结果片段一起放入最终的测试程序，从而创建代码片段及其必要的上下文。此外，该方法还将一些代码附加到语法中，以帮助生成语法正确的程序。

### 3.1.2 属性语法

Hanford[26]的工作提供了一个基本思路，即可以使用扩展语法为测试编译器生成测试程序。后来，Duncan和Hutchison[28]演示了一个使用属性语法的测试程序生成器。

属性语法是通过向语法规则中添加属性来扩展上下文无关语法的另一种方式。属性语法是由Knuth[29]引入的，他扩展了上下文无关语法，通过向某些结果附加附加信息(属性)来表达语义。Knuth的公式有两种类型的属性:合成属性和继承属性。当语法规则表示为解析树时，合成属性的值依赖于子属性的值，而继承属性的属性值依赖于父属性的值。

Duncan和Hutchison[28]提出了一种通用的测试语法，作为如何以及在何处将属性添加到语言的具体上下文无关语法的指南。例如，他们将测试语法应用于Ada的上下文无关语法的子集，并向语法产品添加属性。这些属性是非负整数，它们指导测试程序的生成，因为属性的值决定了在生成过程中将使用哪些语法。他们表示，他们的方法在测试Ada编译器的优化器时是有用的。

Burgess[30]提出了一种同样基于属性语法测试Pascal编译器的方法。后来，Burgess和Saidi[31]将其扩展到检查两个Fortran编译器中的优化错误。

与之前的工作类似，他们通过添加属性来扩展语法，并为语法结果分配权重，并生成自检测试程序。自检测试程序的基本思想是与测试程序一起生成断言，这一思想受到Bird和Munoz[32]的工作的启发。该方法使用许多启发式方法来生成测试程序，以便测试编译器应用已知的优化。测试人员可以控制这种启发式的应用，并可以指定语法结果的权重。

Amodio等人[33]的方法是训练一个循环神经网络来生成测试程序。该模型经过训练，生成符合给定语法的数据，并遵循语言的其他格式良好的属性，例如在使用变量之前定义变量。

### 3.1.3 w语法

w语法是由Adriaan van Wijngaarden为指定ALGOL 68[23]而开发的，它是第一种两级语法。w语法由两个上下文无关的语法组成。其中一种与上下文无关的语法是元语法，用于为第二种语法生成终端符号。元语法可以被认为是泛化上下文无关语法的一种方式。使用一致替换等技术，其中所有出现的非终结符都被替换为相同的展开符号，w语法能够增强上下文敏感性。

Bazzichi和Spadafora[34]使用了一种带参数扩展的上下文无关语法，这实际上是一种w语法。他们把这种新语法称为与上下文无关的参数语法。将参数扩展到一些非终结符语法结果。对于这种特殊情况，参数本身是生成变量的语法。他们的目标是生成有效的程序来测试编译器。与Purdom的工作类似，生成算法倾向于对终端进行短派生，并至少使用一次语法的所有生成。Bazzichi和Spadafora实现了测试PLZ/SYS和Pascal编译器的方法。

以前用于生成测试程序的语法导向方法要么基于完整的语言语法，要么基于完整语法的子集。Zelenov等人[35]提出了一种基于语法模型生成测试程序的替代方法。给定语言语法，他们引入语法转换来生成语言语法的受限模型。该模型包含生成目标语言格式良好的句子所需的最小结果集。这个生成器包含一个迭代器和一个映射器，迭代器生成这样的模型表示，映射器将这些表示映射到有效的语言句子。

Zelenov和Zelenova[36]构建了编译器优化器的模型，并生成针对优化的测试。基本思想是构建由编译器执行的优化模型，然后生成包含优化机会的测试程序。

Lindig[37]提出了另一种直接遍历语法结果的方法。他的方法被称为Quest，使用自定义的类语法程序来生成用于测试C编译器的随机测试程序。生成的目标是测试传递给函数的参数是否被原样接收。每个生成器既可以生成类型，也可以为类型生成值，或者接受其他生成器作为输入。因为测试的目标是测试传递给函数的值是否被原原本本地传递下去，所以测试程序包含全局变量、函数、使用全局变量和对函数的调用。这些函数包含检查接收到的参数是否具有预期值的断言。

## 3.2 语法辅助

语法辅助方法接受语法作为输入，并额外使用一些启发式方法来解决上下文敏感性。这些方法从类似模板的固定代码片段开始，作为占位符，然后利用语法生成程序的其余部分。此外，为了指导整体生成过程，一些方法还将测试驱动文件作为输入，而其他方法则将其添加到语法产生式中。

Sirer等人[38]提出了一种语法辅助方法，用于测试Java虚拟机（JVM），通过概率迭代文法产生式。给定一个语法和一个称为种子的骨架程序，该方法输出自检测测试程序，即带有断言来验证JVM的正确性。输入的语法规范包含需要用其他信息扩展的产生式，例如，特定产生式可使用的次数限制，以及描述产生式适用上下文的保护条件。骨架程序包含有关每个产生式概率的注释，以及由产生式扩展生成的代码片段填充的空白。

Yang等人[9]提出了一种名为Csmith的语法辅助工具，用于创建C测试程序。该工具基于Randprog，并且除了函数、全局和局部变量、常量和易失性变量之外，还生成包含控制流语句、结构体、数组和大多数C表达式的程序。这种方法使用复杂的启发式方法来避免生成具有未定义行为的C程序以及依赖未指定行为的C程序。生成从创建主函数和一组struct类型声明开始，每个struct包含随机数量的随机类型成员变量。使用主函数作为起点，根据C语法的子集生成其余的C代码。根据生成的当前状态，Csmith基于概率表和过滤函数选择一个可允许的产生式。过滤函数强制执行上下文敏感性。在生成过程中，Csmith执行某些安全检查，并仅在所有安全检查通过时创建代码片段。

之后不断有人提出了Csmith测试程序生成器的变体。Morisset等人[39]修改了Csmith，以生成测试C/C++并发错误的程序。他们修改后的版本增加了对互斥锁和原子变量的支持，以及用于锁定和解锁互斥锁变量的系统调用。此外，Lidbury等人[40]基于Csmith构建了一个用于OpenCL编译器的测试程序生成器，名为CLsmith。CLsmith总共有六种模式，在不同的模式下生成不同类型的OpenCL内核（即测试程序）。例如，在VECTOR模式下，CLsmith通过引入生成具有向量类型的变量和表达式的能力，以及执行OpenCL中可用的丰富的向量操作，扩展了Csmith。与通过附加语言功能扩展Csmith不同，Swarm测试限制了生成特定程序所用的语言功能集。例如，Swarm测试可能配置Csmith在一些测试程序中不使用任何数组，并在其他一些程序中不使用任何指针。这种方法的直觉是，通过集中使用语言特性的子集来覆盖一些错误，而不是将测试工作平均分配到所有特性上。稍后，Alipour等人[41]进一步提出了有向Swarm测试，该方法旨在通过调整可用语言特性的集合来生成仅关注编译器部分的测试程序。具体来说，通过分析过去测试结果的统计数据，有向Swarm测试将Csmith配置在一组C语言特性上，以更高的概率生成覆盖编译器特定部分的测试程序。

Holler等人[42]并不像以前的方法那样生成或硬编码占位符代码，而是使用真实世界的程序作为占位符。他们从一个语料库中获取这些程序，并将程序的某些部分替换为随机生成的代码片段。代码片段是通过遍历语法产生式并在特定产生式有多个选择时进行随机决策而生成的。在最大迭代次数之后，剩余的非终结符总是被终结符替换。这些替换终结符再次从大型代码语料库中获取。此外，该方法使用一些启发式方法，例如重命名标识符，以适应生成的代码片段与目标或占位符程序。

所有先前的语法辅助方法都以给定的占位符输入开始。相比之下，Boujarwah等人[43]首先生成一个测试程序，然后生成额外的代码以获得一个语义正确的测试程序。根据一个上下文无关的语法，他们的方法测试Java编译器的某些语义特性。他们首先决定要测试语言的哪些语义特性，然后使用上下文无关的语法，以及一个测试驱动文件来指导生成。测试驱动文件包含有关要生成的语言构造的信息，即要生成每种这种语言构造的数量以及它们将生成的顺序。例如，如果在生成过程中，Bourjarwah等人决定生成用于测试循环的测试程序，则测试驱动文件可能包含应该生成的循环类型（for、while、do-while）以及它们的数量。一旦生成了一个语言构造，该方法就会生成生成的代码所需的上下文。例如，他们生成所需的类型声明或添加必要的导入。对上下文无关的语法的迭代基于之前讨论的Purdom算法[8]。

## 3.3 其他方法

并不是所有自动测试程序生成的方法都是建立在目标编程语言的语法之上的。下面讨论一组不直接使用语法作为输入的方法。它们中的许多基于预定义的代码模板生成测试程序，这些模板为测试程序指定了一个框架，然后填充了额外的代码片段。

Berry[11]提出了基于真实程序员使用语言特性的频率为编译器生成测试程序。为此，他们收集语言特性使用情况的统计数据，然后根据经常使用的特性设计测试程序。该方法背后的见解是，在实际使用过程中，使用频率较高的特性更有可能被编译器使用，因此也更有可能触发bug。该方法根据收集到的统计信息生成不同的硬编码语言片段。

Mandl[12]是一种避免在程序中产生重复元素的方法。它使用一种称为正交拉丁平方的独特方法来生成用于验证Ada编译器的测试程序。拉丁方阵是一种方阵，其中每一行和每一列只包含一个元素一次，即矩阵的所有元素都是唯一的。如果将两个矩阵组合成另一个拉丁方阵，则它们是正交拉丁方阵。Mandl将测试模板表示为正交拉丁方阵，并通过将模板矩阵中的一行元素替换为允许的值来生成测试程序。使用这种拉丁方阵表示有助于创建具有不同配置的测试模板。每个配置可以生成多个独特的测试程序。例如，假设目标是生成算术表达式。给定表达式的操作数，测试模板矩阵可以包含算术表达式的操作符。在使用操作数时遍历模板矩阵的每一行，该方法生成唯一的算术表达式。

第3.2节中介绍的Csmith方法是对非基于语法的Randprog方法的扩展，Randprog方法由Eide和Regehr[10]提出。顾名思义，Randprog生成随机的C程序来查找C的volatile限定符的错误编译。Randprog生成的每个程序都是硬编码的，首先包含一些随机初始化的全局变量，这些全局变量要么是const，要么是volatile。然后程序包含声明局部变量和包含使用全局变量和局部变量的表达式的函数。最后，为了使程序可执行，每个程序都包含一个main函数。

Nagai等人[44]提出了一种类似于Randprog的方法。他们在C编译器中测试算术表达式的代码优化。每个生成的测试程序都包含初始化的全局和局部变量，以及使用随机操作符和变量的算术表达式。

生成使用启发式来避免生成未定义的行为。在生成过程中，该方法预先计算每个表达式的结果，并插入一个运行时检查，将预先计算的结果与实际结果进行比较。在后续工作中，Nagai等人[45]通过改变算术表达式改进了他们的工作，我们将在第3.4节中解释。

Randprog[10]和Nagai等人的方法首先生成固定的语言结构，然后随机生成程序的其余部分。相比之下，Palka等人[46]，类似于一些语法辅助方法，从占位符开始，并用随机的Haskell术语扩展占位符。除了占位符之外，它们还接受用判断命题表示法编写的类型规则作为输入，并生成类型良好的Haskell术语。每个类型规则都有一个关联的环境，其中包含标识符及其相应类型的列表。对于每个类型规则，该方法首先递归地生成前提中出现的项，然后在结果中生成项。Dewey等人[47]没有直接使用类型规则，而是使用约束逻辑规划(CLP)提出了一种模糊测试技术。类型判断以CLP规范编写，并用作CLP引擎生成的输入。它们扩展了群测试的思想，以发现Rust类型检查器中的缺陷;也就是说，他们不是创建一个包含所有Rust的生成器，而是创建不同的生成器，专注于Rust类型系统的不同部分。

Palka等人的方法后来被Midtgaard等人[48]改进。他们注意到，在某些语言中，表达式的求值取决于未指定的求值顺序。例如，如果未指定求值顺序，则表达式从左到右求值的结果可能与从右到左求值的结果不同。

由于很难判断依赖于求值顺序的表达式的正确结果，因此他们的方法试图避免创建这样的表达式。为此，他们对Palka等人的方法进行了改进，引入了类型和效果规则。在生成过程中，该方法使用一些预定义的规则避免了依赖于求值顺序的程序的生成。

有些方法使用用户可以控制的配置或模板文件，而不是在生成器中对生成的测试程序的特性进行硬编码。Austin等人[49]提出了这样一个程序生成器，用于测试Ada编译器。该方法通过对可选语法决策采取随机决策，以递归下降方式生成复杂的Ada表达式。生成器是可配置的，因为测试人员可以指定，例如，生成的整数的大小或用于随机决策的种子。

Yoshikawa等人[50]利用语法知识和启发式的应用，提出了类似的配置驱动生成方法。生成开始于生成随机数的具有非循环父关系的Java类。接下来是为每个类生成字段，然后是生成方法。然后用控制流信息填充这些方法。例如，将方法调用随机添加到每个方法体中。这些添加带有一些约束，以避免某些行为，例如无限方法调用。

Zhao等人提出了另一种以测试编译器优化为目标的配置驱动的测试程序生成方法。该方法采用测试配置文件作为输入，该文件指定，例如，应该生成哪种变量类型，需要生成哪些操作符，以及在测试程序中使用多少分支和循环。此外，用户可以配置要测试的优化。要测试的优化被指定为时间逻辑公式，然后将其转换为图结构。然后将图形转换为模板，其展开导致生成的程序。

与Berry类似，Ching和Katz[52]提出了一种使用从实际应用中收集的程序来测试APL编译器的方法。此外，Ching和Katz也提出从模板生成单元测试。这里的模板用表示函数和数据类型的特殊符号表示测试。在生成过程中，这些符号被替换为具体的值和被测试的特定功能。这里的函数是APL的内置函数。

除了模板之外，Kalinov等人[53]的方法还采用输入表达式来生成测试程序，他们使用该表达式来查找并行语言编译器mpC中的错误。模板包含一组mpC操作符，而表达式是由测试器提供的称为种子表达式的有效表达式。从这个种子表达式开始，该方法通过使用种子表达式中的操作数和模板中的操作符来生成多个变体表达式。mpC语言被指定为一种称为蒙太奇的视觉形式[54]，它可以表达语法和执行行为。Kalinov等使用该规范在测试程序生成过程中过滤出正例和负例[53]。

Zhang等[55]提出了骨架程序枚举。给定一个程序框架，也就是说，源代码的漏洞要用变量名填充，该方法详尽地枚举所有可能的变量使用模式。由于变量名的不同孔赋值可能是等价的，因此该方法在alpha重命名下(即在变量名的一致重命名下)只枚举一组等价程序中的一个程序。

随着最近在软件工件(例如源代码)上使用机器学习的趋势，已经提出了几种基于学习的测试程序生成方法。它们都有一个共同的基本思想，即从代码示例的语料库中学习模型，然后使用该模型生成额外的测试程序。这些方法的不同之处在于它们所使用的模型的种类和它们所针对的程序属性的种类，如下所述。我们已经在3.2节中介绍了Amodio等人的一种方法，他们训练一个循环神经网络来生成测试程序。另一种名为TreeFuzz[56]的基于学习的方法学习一组树形数据的概率生成模型，例如由AST表示的程序。这些模型处理程序的语法和语义属性，例如，通过学习特定节点通常具有的子节点或通过学习相同变量出现之间的类似定义使用的关系。该方法已用于测试JavaScript引擎。

Bastani等人[57]提出基于语法接受的数据示例和对语法解析器的黑盒访问来学习语法。该方法迭代地构建一个接受日益通用的语言的语法，从合成正则表达式开始，然后将正则表达式泛化为与上下文无关的语法。一旦掌握了语法，就可以用来测试新的测试程序。该方法已用于各种数据格式，包括Python、Ruby和JavaScript中的测试程序。

## 3.4 本章小结

本章主要详细介绍了常见的三种测试程序生成技术的方向，包括语法导向，语法辅助和其他方法，以及常见的使用这些方法的测试程序生成器。

# 4 SolSmith的实现

本章主要介绍了SolSmith生成代码的主要流程和生成过程中为了能够高效的发现Solidity编译器问题所设计的生成策略。

## 4.1 生成代码

本节主要介绍生成测试程序的流程以及流程中每个节点具体进行的任务。

### 4.1.1 创建环境

首先SolSmith分别生成一组状态变量，一组storage类型的动态数组，一组constant变量，一个修饰器和一个event语句。

状态变量的类型为uint256或int256，在定义时会赋给状态变量一个随机的常数值，同时将这些变量的变量名和变量类型存储起来。数组固定为uint256类型，长度随机，同时会以随机值进行初始化。数组的数组名和初始长度会存储起来。上述状态变量和数组在之后生成的代码中和每个函数的局部变量组成不同的赋值语句，判断条件等内容。

修饰器中会有一些只包含状态变量和数组的算术表达式和一个占位符。修饰器之后会在每个函数上随机安排（可能所有函数都没有使用修饰器）。

event语句中包含所有的状态变量，之后在每个函数的最后，会调用event语句，将状态变量的值记录下来，使用hardhat框架可以方便的获得每个状态变量的值，之后比较不同优化条件下的同一个程序的状态变量的值，就可以得到程序运行过程中编译器是否出现了问题。

在Solidity中变量按作用域划分有三种，分别是状态变量（state variable），局部变量（local variable）和全局变量（global variable）。状态变量是数据存储在区块链上的变量，所有合约内的函数都可以访问，gas消耗高。状态变量在合约内、函数外声明。状态变量类似于C++语言中的全局变量。

### 4.1.2 创建函数框架

每次运行SolSmith都会生成一个包含三个函数的智能合约。每个函数都会随机确定是否有返回值、返回值的类型和是否有修饰器。每个函数都是自顶向下生成的：一个测试函数由单个顶级块组成，其中包含一个语句列表。每个语句执行以下操作之一：

声明一个随机选择的整数类型的新局部变量，并赋给它一个随机常数值。

将一个随机的算术表达式的结果赋值给一个已存在的变量。

开始一个新的条件语句。

开始一个新的循环语句。

开始一个新的内联汇编语句。

对数组的操作。

### 4.1.3 创建语句

SolSmith中的语句主要包含赋值语句，条件语句，循环语句，内联汇编语句。

赋值语句。赋值语句中用的变量主要包括状态变量，数组，局部变量。首先确定等号左边的值。等号右边是一个表达式。该表达式中的节点是下列中的任意一个：二元或三元操作符，强制类型转换，数据元素：局部变量，状态变量，数组，随机常数。每个表达式会限制节点数量，在到达最大节点数量时会结束表达式的生成。

条件语句。每个判断条件都是由一组或多组整型变量的比较构成，多组整型变量的比较之间由逻辑运算符或比较运算符连接。其中在每组整型变量由两个整型变量和一个比较运算符组成，整型变量由局部变量，状态变量，数组组成。使用的比较运算符主要有>，<，==，!=。多组整型变量之间使用的逻辑运算符和比较运算符有：||，&&，==，!=。

循环语句。对于循环语句，SolSmith会限制嵌套的循环深度，同时会将用于

迭代计数器的作用域限制在循环语句中。这样在同一深度的循环语句可以使用同名的迭代计数器来控制循环次数。循环语句主要有两种形式，第一种如图4-1所示，迭代计数器初始化表达式在for循环头中。第二种如图4-2所示，迭代计数器的初始化表达式在for语句前，但是会使用一个括号将初始化语句和整个for循环包括起来，这样可以将迭代计数器的作用域限制在这个循环语句中。循环条件固定为迭代计数器小于一个随机常数。迭代表达式固定为迭代计数器自增。

内联汇编语句。在Solidity中用于内联汇编的语言称为Yul。一个内联汇编块由assembly{代码块}，其中大括号内的代码是Yul语言的代码。每个内联汇编块都类似于一个函数，其中可以由赋值语句，条件语句和循环语句。不过因为语法不同，所以在语句的创建上稍有不同，如图4-3，其中是内联汇编代码块中条件语句和循环语句的形式及例子。

图4-1 循环语句第一种形式

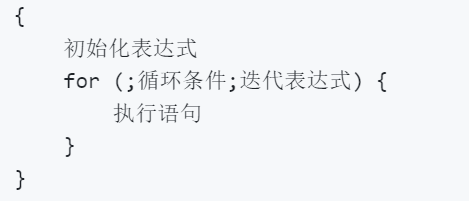
其中只需要对条件语句的条件部分和循环语句的循环头做出部分修改即可。

图4-2 循环语句第二种形式

其中条件语句和循环语句都可以在其中的执行语句块中嵌套赋值语句，条件语句，循环语句和内联汇编语句，可以设定可以嵌套的总的深度，同时可以单独设置条件语句的最大深度和循环语句的最大深度。

在每个函数的最后，如果函数有return语句则在return语句之前都会调用之前定义的event语句用来在测试套件中输出状态变量的值。之后将会对比在不同优化条件下的同一个程序的状态变量的值。只比较状态变量是因为，只有状态变量会将值保存在区块链上，其他的局部变量在变量在程序运行结束后不会有任何影响。

## 4.2 生成策略

### 4.2.1 频繁启用优化

根据JOHN REGEHR[27]等人的研究，当从相同分布中抽取许多随机选择时，随机生成的测试用例最终看起来都有些相似。这种现象减少了生成代码的多样性，从长远来看，会导致测试效率降低。

对于SolSmith，我们应用了类似于JOHN REGEHR等提出的方法，系统地扭曲概率分布，以导致某些优化更频繁地启动。在生成的代码的随机选择的子区域中应用一些分布变化。由于不同的生成策略通常会组合在一起，SolSmith允许它们任意重叠，从而在代码生成中带来额外的多样性。

这些策略是从我们对编译器优化器如何工作，以及编译器过去在哪些方面出现过问题或容易出现问题得出的。

内联汇编中内存的优化问题。Yul优化器会考虑在最外层内联汇编块中的所有内存写入，如果这些写入从未被读取，则视为未使用并将其删除。这在该内联汇编块是整个Yul程序的情况下是有效的，而这总是通过新的via-IR流水线生成的内联汇编代码的情况。当使用该流水线时，内联汇编块永远不会被单独优化。相反，它们将作为整个Yul输入的一部分进行优化。

然而，如果内联汇编块不引用周围Solidity代码中定义的任何变量，那么传统的代码生成过程在内联汇编块上单独运行Yul优化器。因此，如果在同一汇编块中从未读取过写入的内存，则这些内存写入也将被删除，即使稍后可能通过后续内联汇编块访问写入的内存。

传统代码生成根本不会在访问Solidity变量的内联汇编块上运行Yul优化器，这显著减少了受影响的情况数量。大多数内联汇编块要么从周围的Solidity代码中读取或写入值，要么是完全独立的，或者接管程序流程直至程序结束，在实践中很难发生错误。因此，我们在特意在内联汇编中引入大量的Solidity中的局部变量和状态变量，来触发Yul优化器，来更容易发现其中的问题。

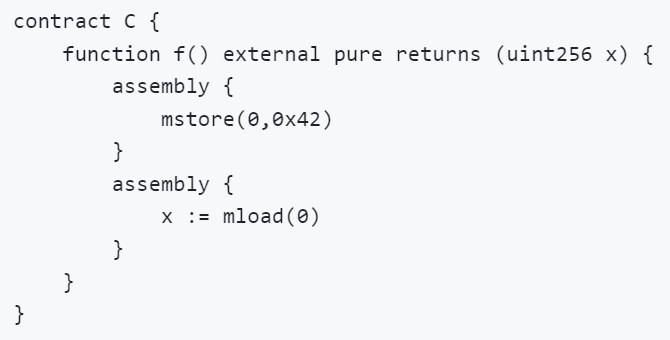
如图4-3中的代码所示，如果使用不启用优化器的传统代码生成方式，那么在第一个内联汇编代码块中的内容不会被删除，x能够被正确的赋值，函数能够得到预想的结果。如果使用启用优化器的代码生成方式，在执行第二个内联汇编代码块的时候，因为在第一个内联汇编代码块中写入的内存从未被使用过，因此优化器会将删除已经写入的内容，函数f最终的返回值为0。

图4-3 内联汇编中内存优化问题代码示例

复制脏数据字节数组到存储中导致的问题。最初认为，只有使用内联汇编才能观察到存储中的脏数据。然而，通过使用空的push()来调整字节数组的大小，而实际上并未向其写入值，就可以在没有使用内联汇编的情况下暴露脏字节。

从内存或calldata复制字节数组到存储的大多数实例都可能受到影响，因为它们可能写入脏数据。只有在对字节数组进行空的push()（在早期版本中还包括对其length字段的赋值）并且假定生成的元素实际上没有被写入时，这些存储中的脏数据才会变得可见。

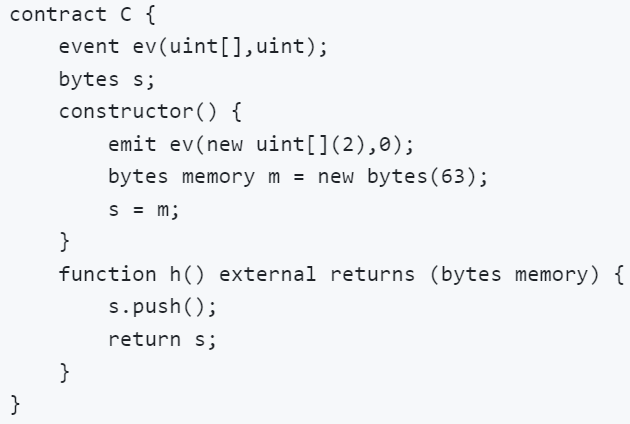
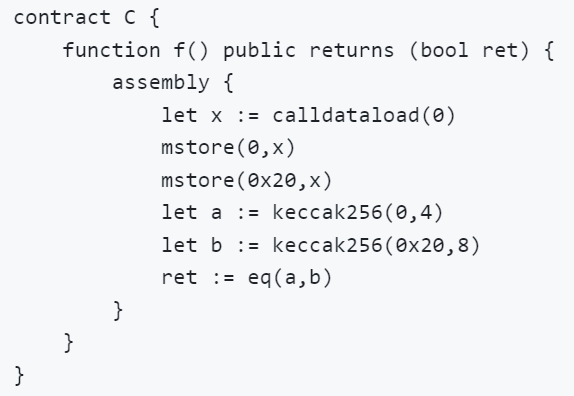
例如图4-4中的代码所示，其中event触发涉及在当前自由内存指针位置写入临时内存。其他一些操作（例如某些keccak256调用）也会以类似的方式使用临时内存。在这种特殊情况下，传递数组的长度将被精确地写入临时内存，以确保在以下分配的63字节后的字节为0x02。然后，在赋值期间，这个脏字节将被写入存储，并在“h”中的push操作中变得可见。

图4-4 复制脏数据问题代码示例

Keccake256哈希值的错误重用。Keccake256函数一共有两个参数第一个参数m是一个地址，第二个参数是一个非负整数p表示需要读取的数据的长度，该函数的作用是计算从地址m开始到m+p-1为止的数据的哈希值。如果keccake256函数在编译时就能够得知要读取的内存的内容，在这种情况下，即使keccake256的第二个参数即所需计算的内容长度不同，但是最终的Keccake256哈希值却是相同的。在如图4-5所示的代码中，哈希无法在编译时计算但优化器仍将它们视为相等。

因此我们会在代码生成过程中有一定倾向的在内联汇编代码中加入keccake256函数，以便能够更多地启用Solidity编译器中有关的优化器，从而发现更多的相关方面的编译器错误。

图4-5 Keccak256哈希值计算错误代码示例

### 4.2.2 避免中间表征导致的区别

Solidity 可以通过两种不同的方式生成 EVM 字节码： 要么直接从 Solidity 到 EVM 操作码（“旧编码”）， 要么通过在 Yul 中的中间表示法（“IR”）（“新编码” 或 “基于IR的编码”）。

引入基于 IR 的代码生成器的目的是，不仅使代码生成更加透明和可审计， 而且能够实现更强大的跨函数的优化通道。

由于一些原因，旧的和基于 IR 的代码生成器之间存在着微小的语义差异， 主要是在那些我们无论如何也不会期望人们依赖这种行为的领域。 本节强调了旧的和基于IR的代码生成器之间的主要区别。

下面的一些内容主要是为了避免在使用新旧不同的代码生成器导致出现的问题。

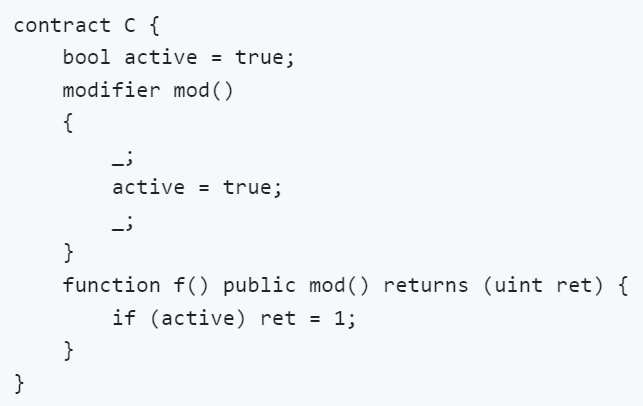
占位符多次使用。如果占位符“\_;”在一个修饰器中多次使用会导致意料之外的结果。在旧的代码生成器中，每个函数参数含返回变量在堆栈中都有一个固定的槽。如果因为多次使用占位符而使函数多次运行，或在一个循环中使用，那么函数参数或返回值的变化在下一次执行中是可见的。新的代码生成器使用实际的函数来实现修饰器，并将函数参数传递下去。这意味着对一个函数主体的多次使用将得到相同的参数值，而对返回变量的影响是，它们在每次执行是都会被重置为默认值（零）。

图4-6 占位符多次使用导致的问题代码示例

在如图4-6所示的代码中，如果使用的是旧的代码生成器，作为返回变量的ret在第一次占位符使用前只被初始化为零，然后使用第一次占位符后ret修改为1。在第二次使用占位符时，ret没有被再次初始化，且第二次没有再次修改ret的值（因为active修改为了false），因此ret会保持第一个值。所以使用旧的代码生成器的结果为1。而对于新的代码修改器，ret作为返回参数，在每次使用占位符前都会重新初始化，所以返回结果为0。

为了避免出现这种因为使用新旧代码生成器的区别而导致的变化，所以要在生成的代码中避免这种行为。我们采用的方式是在生成修饰器的同时定义一个计数器并初始化为0，当在某行随机生成了一个占位符后，计数器加一。生成每行的语句时，要判断计数器是否为零，如果为零则说明在此之前没有生成过占位符，说明这行可以生成占位符。如果占位符不为零，则说明之前有过占位符，那么这行不能生成占位符，会重新随机选择生成的语句直到不再是占位符为止。

表达式评估顺序问题。对于旧的代码生成器，表达式的评估顺序是没有规定的。对于新的代码生成器，会试图按照从左到右的顺序进行评估，但是新的代码生成器只是会试图按照这个顺序，并不保证一定按照这个顺序。这点会导致最终的结果不同。

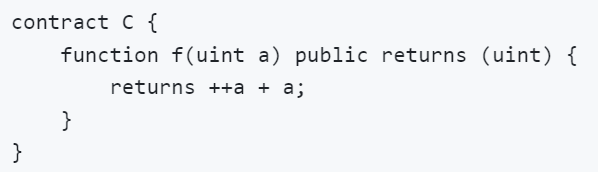
例如对于图4-7示的代码，对于f(1),如果使用的是旧的代码生成器，因为旧的代码生成器没有规定表达式评估顺序，所以可能会先处理++a再处理+a即为2+2，返回结果为4，也有可能先处理后一个a，再处理++a，即为2+1，返回结果为3。如果使用的是新的代码生成器，一般会先处理++a再处理+a，结果为2+2=4。但是新的代码生成器也不保证一定会返回4。

图4-7 表达式评估顺序导致的问题代码示例

为零避免出现这种情况，我们在生成每一条算术表达式的时候，会使用一个哈希表记录当前使用过的变量名称，我们会先选定当前位置使用的变量，然后再随机在该位置添加自增自减操作，如果是单独使用，则哈希表中对应的值为1，如果是有自增自减操作，则哈希表中对应的值为2。如果再之后的位置再次选中该变量，则进行判断该变量在哈希表中对应的值允许进行何种操作，执行允许的操作。如果该变量在开始时是单独使用，则之后依旧可以单独使用。但如果该变量在开始时是自增操作，则再次选中该变量时，会随机生成一个常数来代替该变量。

## 4.3 本章小结

本章主要介绍了SolSmith是如何实现的，以及我们使用何种策略来使得SolSmith能够生成更有效的测试程序。其中第一节主要介绍了SolSmith生成测试程序的整体流程，以及每个流程具体的内容。第二节主要介绍了我们为了使SolSmith生成的程序能够更有效的用于测试编译器，为此做的工作，主要包括认为扭曲部分代码生成概率，利用Solidity的中间表征的相关内容。

# 5 实验结果与分析

在上文中已经详细介绍了SolSmith的主要思想和随机生成程序的基本流程。为了验证随机生成程序对编译器测试的有效性。本章主要对SolSmith对编译器检验错误的能力和实验结果进行分析介绍。本章首先会介绍实验流程设计，包括实验环境，实验对象以及实验步骤。之后会对实验结果进行分析。

## 5.1实验环境

本文进行实验的环境配置及使用工具参数如表4-1所示

表4-1 环境参数

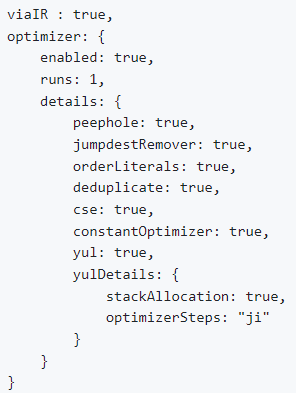
|  |  |
| --- | --- |
| 配置名称 | 具体参数 |
| CPU | 13th Gen Intel(R) Core(TM) i7-13700F |
| 内存 | 32G |
| 操作平台 | Ubuntu 22.04 |
| Solidity编译器版本 | 0.8.0 |
| 测试框架 | Hardhat 2.12.2 |
| Python平台 | Pycharm-community-2023.3.3 |
| C++平台 | Visual Studio 2022 |

其中Hardhat是一个用于以太坊平台的软件的开发环境。它由编辑、编译、调试和部署智能合约和分布式应用的不同组件组成，所有这些组件共同组成一个完整的开发环境。

## 5.2 实验过程

实验主要是将SolSmith生成的程序在同一个Solidity编译器启用不同优化设置下运行。然后将生成的测试程序分别放入不同的配置下运行，之后比较在不同配置下运行测试程序得到的结果是否相同。

在实验过程中我们使用的编译器设置主要有如图5-1所示的内容。

 图5-1 编译器可启用配置

其中每个选项代表的含义分别如下：

viaIR表示改变编译管道以通过Yul中间表示法。这个选项再默认情况下是false，即关闭的。

optimizer表示优化器设置。优化器默认情况下是禁用的。

Enabled选项控制编译器的启用与否。将enabled设置为false，实际上仍然保留了一部分优化功能，而省略enabled实际上会禁用所有优化。因此将enabled设置为false和省略该选项，实际上并不相等。为了能够一次启用和关闭所有优化，我们在修改编译器配置时会直接将整个optimizer内容进行注释处理。

Runs选项会根据预期的运行代码次数进行优化，较低的值将更多地针对初始部署成本进行优化，较高的值将更多地针对高频使用进行优化。最小值为1，最大值为2^32-1。

details控制打开或关闭优化器组件的细节。如果不启用details，部分优化器默认是启用的，想要关闭它们需要给出details，并在其中关闭，在介绍详细的优化器部件时会给出说明。

peephole控制窥视孔优化的开关，窥视孔优化器默认是打开的，需要使用 details 来关闭它。

jumpdestRemover控制跳板移除器的开关。默认是打开的，需要使用details来关闭它。

orderLiterals在换元运算中，有时会对字词重新排序。

deduplicate移除重复的代码块。

cse常见的子表达式消除，这是最复杂的步骤，但也能提供最大的收益。 constantOptimizer优化代码中字面数字和字符串的表示；

yul新的Yul优化器。主要在ABI coder v2 和 内联汇编的代码上运行。它与全局优化器设置一起被激活，并且可以在这里停用。

yulDetails Yul优化器的调优选项。只有在yul选项设置为true时才可以使用yulDetails。

stackAllocation改善变量的堆栈槽的分配，可以提前释放堆栈槽。如果Yul优化器被激活，则默认激活。

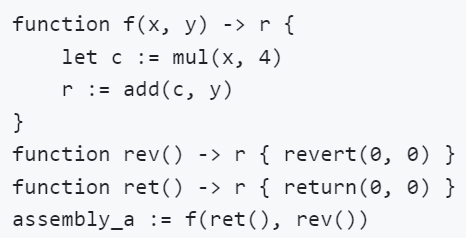
optimizerSteps选项用于控制要应用的优化步骤，可以同时控制优化序列和清理序列。两个序列之间以“:”分隔，即以 优化序列:清理序列的形式给出。该选项是可选的，如果不提供，优化序列和清理序列都会使用默认序列。如果只提供了一个序列，另一个将不会被运行。如果只提供了分隔符“:”，那么优化序列和清理序列都不会执行，如果设置为空值，则只使用默认的清理序列，不应用任何优化步骤。

我们的实验配置选取主要有以下几种，第一种是不启用任何优化，为标准配置；第二种是启用所有优化并且优化的方向是针对初始部署成本进行优化，即rus属性的值尽可能小，我们设置其为1；第三种是启用所有优化并且优化方向是针对高频使用来进行优化，即runs属性的值尽可能大，我们设置其为2^32-1。此外还会根据Solidity编译器已知的问题调整优化器启用顺序和不同的优化器开关组合，来测试更多的配置方案。

## 5.3 实验结果

我们一共测试了10万条生成的测试程序，与标准配置相比较共有两类配置出现问题，其中运行结果如下：

我们一共发现了Solidity编译器的三个问题。

第一种问题是在特定编译器配置下运行测试程序时出现崩溃的现象。该错误需要在特定的优化器顺序下才能触发。其触发代码示例如图5-2所示。其触发配置如图5-3所示。

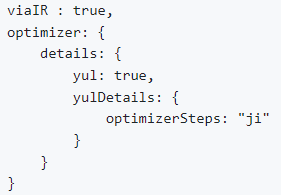
图5-2 第一种问题代码示例

图5-3 第一种问题具体配置

该错误需要在编译器在通过Yul生成EVM码的前提下，使用特定的优化器顺序才能够正确触发。该错误本质上是在特定的优化器顺序下，导致函数参数读取顺序不一致的问题。如图5-3所示的代码，在正常不启用优化器时，Solidity编译器读取f函数的参数顺序时从右往左的。但是在当优化器以特定的顺序启用，Solidity编译器读取f函数的参数顺序将会从左向右。最终导致在两种不同配置下程序运行结果不同。

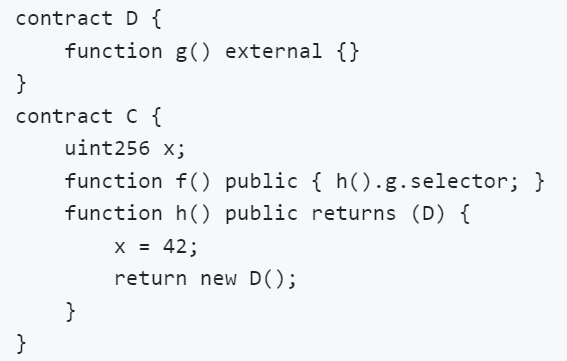
第二种问题是在不同智能合约之间调用函数时出现的问题。代码示例如图5-4所示。

图5-4第二种问题代码示例

该错误需要存在两个智能合约，其中第一个智能合约D存在一个为可见性为external的函数，该类型函数只能被当前智能合约外的智能合约调用。同时在第二个智能合约C中有一个函数h将第一个智能合约实例化，同时在第二个智能合约C中有另一个函数f通过函数h产生的智能合约D的实例调用D中的函数g。在这种情况下，只需要将viaIR选项设置为true，就会产生与标准配置不同的结果。这是由于传统代码生成流程存在问题导致的，如果使用通过IR的代码生成流程，x将会被赋值为42并产生智能合约D的实例，但是如果使用传统的代码生成流程将导致x为0，同时不会产生智能合约D的实例。

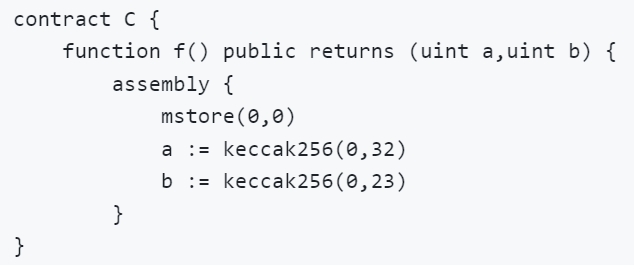
第三种问题是在计算keccak256哈希值出现的。代码示例如图5-6所示。

图5-6 第三种问题代码示例

出现这种错误需要将cse选项设置为true。该错误出现的原因是如果调用了两次keccak256函数，并且两次调用的地址是相同的，那么将keccak256函数的第二个参数如果往上取值到32的倍数，如果两次调用中经过上述操作得到的值是同一个，那么这两次调用的返回值是相同的，即使第二个参数值不相同。

此外我们在编译器测试过程中我们还发现启用优化的两种设置的编译器，runs属性无论是设置为1还是设置为2^32-1，没有出现不同的运行结果。可能是由于我们生成的测试程序对于Solidity编译器来说，不论是向更节省部署成本的方向优化还是向更节省执行成本的优化结果都是一样的。

在所有的10万条测试程序中，一共有5000多条测试用例出现了不同结果。其中出现的问题全部（或大部分问题）为内联汇编代码在优化时出现的问题。

## 5.4 本章小结

本章主要对我们编写的SolSmith进行了实验测试和实验结果分析。其中5.1节主要介绍了实验所使用的实验环境，5.2节介绍了不同配置的编译器分别设置了什么属性，以及各属性代表的含义5。3节介绍了实验结果，以及对实验结果进行分析。

# 结论

本文设计并实现了一个用于测试Solidity编译器的测试程序生成工具SolSmith。该工具的基本思想是使用语法辅助的测试程序生成技术，同时我们依照JOHN REGEHR等人的发现，在SolSmith 中将能够触发Solidity编译器优化的部分代码人为的提高出现的频率，以便能够更高效的调用优化器，来更有效的发现相关的优化器问题。同时也生成测试程序时注意到相关的Solidity特性，来避免不必要的问题。我们使用SolSmith从Solidity编译器中成功发现了Solidity编译器的3个问题。根据实验结果可知我们设计的SolSmith生成的测试用例能够有效的进行Solidity编译器测试。

当然该工具目前还存在一些不足。目前SolSmith中的变量只集中在整型方向，缺乏布尔、字节等类型的变量。同时为了测试方便，SolSmith生成的函数中全都是不包含参数的，之后可以在生成函数是加入参数。之后的改进还可以加入更多的不同智能合约以及不同的智能合约之间的关系可以更复杂，如可以加入继承等内容。

# 参考文献

1. Szabo N. Formalizing and securing relationships on public networks[J]. First monday, 1997.
2. Clack C D. Smart Contract Templates: legal semantics and code validation[J]. Journal of Digital Banking, 2018, 2(4): 338-352.
3. Clack C D, Bakshi V A, Braine L. Smart contract templates: foundations, design landscape and research directions[J]. arXiv preprint arXiv:1608.00771, 2016.
4. Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum project yellow paper, 2014, 151(2014): 1-32.
5. Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs[C]. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014.
6. Bauer S, Cuoq P, Regehr J. Deniable backdoors using compiler bugs[J]. International Journal of PoC|| GTFO, 2015, 0x08: 7-9.
7. Chen J, Hu W, Hao D, et al. An empirical comparison of compiler testing techniques[C]. Proceedings of the 38th International Conference on Software Engineering. 2016: 180-190.
8. Purdom P. A sentence generator for testing parsers[J]. BIT Numerical Mathematics, 1972, 12: 366-375.
9. Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers[C]. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011: 283-294.
10. Eide E, Regehr J. Volatiles are miscompiled, and what to do about it[C]. Proceedings of the 8th ACM international conference on Embedded software. 2008: 255-264.
11. Berry D M. A new methodology for generating test cases for a programming language compiler[J]. ACM Sigplan Notices, 1983, 18(2): 46-56.
12. Mandl R. Orthogonal Latin squares: an application of experiment design to compiler testing[J]. Communications of the ACM, 1985, 28(10): 1054-1058.
13. Mitropoulos C, Sotiropoulos T, Ioannidis S, et al. Syntax-Aware Mutation for Testing the Solidity Compiler[C]//European Symposium on Research in Computer Security. Cham: Springer Nature Switzerland, 2023: 327-347.
14. Chen J, Patra J, Pradel M, et al. A survey of compiler testing[J]. ACM Computing Surveys (CSUR), 2020, 53(1): 1-36.
15. McKeeman W M. Differential testing for software[J]. Digital Technical Journal, 1998, 10(1): 100-107.
16. Chen Y, Groce A, Zhang C, et al. Taming compiler fuzzers[C]. Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. 2013: 197-208.
17. Wang X, Chen H, Cheung A, et al. Undefined behavior: what happened to my code?[C]. Proceedings of the Asia-Pacific Workshop on Systems. 2012: 1-7.
18. Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 2: Instruction Set Reference. Intel, March 2012.
19. Bloch J, Gafter N. Java puzzlers: traps, pitfalls, and corner cases[M]. Pearson Education, 2005.
20. Lindig C. Random testing of C calling conventions[C]. Proceedings of the sixth international symposium on Automated analysis-driven debugging. 2005: 3-12.
21. Nagai E, Hashimoto A, Ishiura N. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers[C]. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013). 2013: 88-93.
22. Chen T Y, Cheung S C, Yiu S M. Metamorphic testing: a new approach for generating next test cases[J]. arXiv preprint arXiv:2002.12543, 2020.
23. A.vanWijngaarden A. Orthogonal design and description of a formal language[J]. Stichting Mathematisch Centrum. Rekenafdeling, 1965 (MR 76/65).
24. Meertens L, Koster C H A. Basic English, a generative grammar for a part of English[C]//Euratom Seminar\Machine en Talen", Amsterdam. 1962.
25. Koster C H A. Affix grammars for programming languages[C]//International Summer School on Attribute Grammars, Applications, and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991: 358-373.
26. Hanford K V. Automatic generation of test cases[J]. IBM Systems Journal, 1970, 9(4): 242-257.
27. Houssais B. Verification of an algol 68 implementation[C]//Proceedings of the Strathclyde ALGOL 68 conference. 1977: 117-128.
28. Duncan A G, Hutchison J S. Using attributed grammars to test designs and implementations[C]//Proceedings of the 5th international conference on Software engineering. 1981: 170-178.
29. Knuth D E. Semantics of context-free languages[J]. Mathematical systems theory, 1968, 2(2): 127-145.
30. Burgess C J. Towards the automatic generation of executable programs to test a Pascal compiler[J]. Software Engineering, 1986, 86: 304-316.
31. Burgess C J, Saidi M. The automatic generation of test cases for optimizing Fortran compilers[J]. Information and Software Technology, 1996, 38(2): 111-119.
32. Bird D L, Munoz C U. Automatic generation of random self-checking test cases[J]. IBM systems journal, 1983, 22(3): 229-245.
33. Amodio M, Chaudhuri S, Reps T W. Neural attribute machines for program generation[J]. arXiv preprint arXiv:1705.09231, 2017.
34. Bazzichi F, Spadafora I. An automatic generator for compiler testing[J]. IEEE Transactions on Software Engineering, 1982 (4): 343-353.
35. Zelenov S V, Zelenova S A, Kossatchev A S, et al. Test generation for compilers and other formal text processors[J]. Programming and Computer Software, 2003, 29(2): 104-111.
36. Zelenov S, Zelenova S. Model-based testing of optimizing compilers[C]//International Workshop on Formal Approaches to Software Testing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007: 365-377.
37. Lindig C. Find a compiler bug in 5 minutes[C]//Proc. ACM Intl. Symposium on Automated Analysis-Driven Debugging. 2005: 3-12.
38. Sirer E G, Bershad B N. Using production grammars in software testing[J]. ACM SIGPLAN Notices, 1999, 35(1): 1-13.
39. Morisset R, Pawan P, Zappa Nardelli F. Compiler testing via a theory of sound optimisations in the C11/C++ 11 memory model[J]. ACM SIGPLAN Notices, 2013, 48(6): 187-196.
40. Lidbury C, Lascu A, Chong N, et al. Many-core compiler fuzzing[J]. ACM SIGPLAN Notices, 2015, 50(6): 65-76.
41. Alipour M A, Groce A, Gopinath R, et al. Generating focused random tests using directed swarm testing[C]//Proceedings of the 25th International Symposium on Software Testing and Analysis. 2016: 70-81.
42. Holler C, Herzig K, Zeller A. Fuzzing with code fragments[C]//21st USENIX Security Symposium (USENIX Security 12). 2012: 445-458.
43. Boujarwah A S, Saleh K, Al-Dallal J. Testing syntax and semantic coverage of Java language compilers[J]. Information and Software Technology, 1999, 41(1): 15-28.
44. Nagai E, Awazu H, Ishiura N, et al. Random testing of C compilers targeting arithmetic optimization[C]//Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012). 2012: 48-53.
45. Nagai E, Hashimoto A, Ishiura N. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions[J]. IPSJ Transactions on System LSI Design Methodology, 2014, 7: 91-100.
46. Pałka M H, Claessen K, Russo A, et al. Testing an optimising compiler by generating random lambda terms[C]//Proceedings of the 6th International Workshop on Automation of Software Test. 2011: 91-97.
47. Dewey K, Roesch J, Hardekopf B. Fuzzing the Rust typechecker using CLP (T)[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015: 482-493.
48. Midtgaard J, Justesen M N, Kasting P, et al. Effect-driven QuickChecking of compilers[J]. Proceedings of the ACM on Programming Languages, 2017, 1(ICFP): 1-23.
49. Austin S M, Wilkins D R, Wichmann B A. An Ada program test generator[C]//Proceedings of the conference on TRI-Ada'91: today's accomplishments; tomorrow's expectations. 1991: 320-325.
50. Yoshikawa T, Shimura K, Ozawa T. Random program generator for Java JIT compiler test system[C]//Third International Conference on Quality Software, 2003. Proceedings. IEEE, 2003: 20-23.
51. Zhao C, Xue Y, Tao Q, et al. Automated test program generation for an industrial optimizing compiler[C]//2009 ICSE Workshop on Automation of Software Test. IEEE, 2009: 36-43.
52. Ching W M, Katz A. The testing of an APL compiler[J]. ACM SIGAPL APL Quote Quad, 1993, 24(1): 55-62.
53. Kalinov A, Kossatchev A, Petrenko A, et al. Coverage-driven automated compiler test suite generation[J]. Electronic Notes in Theoretical Computer Science, 2003, 82(3): 500-514.
54. Kutter P W, Pierantonio A. Montages Specifications of Realistic Programming Languages[J]. J. Univers. Comput. Sci., 1997, 3(5): 416-442.
55. Zhang Q, Sun C, Su Z. Skeletal program enumeration for rigorous compiler testing[C]//Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. 2017: 347-361.
56. Patra J, Pradel M. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data[J]. TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664, 2016.
57. Bastani O, Sharma R, Aiken A, et al. Synthesizing program input grammars[J]. ACM SIGPLAN Notices, 2017, 52(6): 95-110.

# 致谢

弹指一挥间，研究生的三年时光就要结束了，到今天我还记得第一次来实验室，第一次开组会的场景。这三年来我在山东大学，收获了很多，也成长了很多，还有几个月就要告别学生时代，步入社会了，回首这三年来的点点滴滴，不由得感慨良多。

首先要感谢三年来一直对我进行指导的余仲星老师，余老师是和蔼可亲、平易近人的，无论是学术科研上有所困惑，还是日常生活中感到迷惘，余老师总会帮助我，为我指点迷津，可以说余老师是亦师亦友的存在。虽然以后要步入工作岗位，但老师对我的谆谆教导，会永远铭记在我心间，鼓励我在未来的工作中迎难而上、一往无前、争创佳绩。

余老师无论是备课教学，还是探讨学术，亦或是主持项目，始终保持着一丝不苟的负责精神和严谨认真的处事态度。余老师渊博的学识，清晰的谈吐，和蔼的形象，都给我留下了深刻印象，我会在未来的工作生活中向余老师看齐，争取变得更加优秀。

感谢嵌入式实验室的各位老师，在这三年的学习科研生活中帮助了我很多，非常感谢各位老师的指导。此外，我还要感谢各位学长学姐的照顾和帮助，祝你们今后都能在自己的人生道路上一帆风顺、事业有成。学弟学妹们也为我平日里的时光增添了许多乐趣，为实验室带来了许多欢声笑语，希望你们今后在学业和学术上再创辉煌。

感谢陪我走过这三年的朋友和同学，是你们在低谷时陪伴鼓励我，帮助我走出阴霾，在成功时一起分享喜悦，激励我继续前行，正是因为有了你们的存在，才让我一路走来不觉得孤单。

感谢我的父母总是在背后默默地支持我，每一次回家都会为我准备可口的饭菜，整洁的衣物。谢谢你们二十多年来对我的养育之恩，让我在一个温暖的家庭里健康成长，今后我一定会努力工作，尽到我作为子女应尽的义务。

最后我还要感谢审稿的老师，感谢您在百忙之中审阅我的论文。

“长风破浪会有时，直挂云帆济沧海”，二十多年的学生时代即将画上句号，我会坚定理想信念，在日后的工作岗位上发光发热，实现自己的人生价值，无悔这一生。

**攻读学位期间发表的学术论文目录**

Li L, Liang Y, Liu Z, et al. Understanding Solidity Event Logging Practices in the Wild[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023: 300-312.